NTNU – Trondheim
Norwegian University of
Science and Technology

A numerical solver for the advection equation

Jan.Christian.Meyer@ntnu.no

Our objective

- Numerically integrating various functions accounts for *a lot* of applications in parallel computing
  - It easily grows to run for a long time with complicated problems
- In order to look at our various programming models, we need something to parallelize
  - An example problem with a bit of integration fits well
  - If we go through how it works now, we can use it without repeating what it's for later on
- TDT4200 is not a math class
  - We'll use almost the simplest problem there is
  - It's still necessary to understand how it works, though

Advection and diffusion

- Many distributions of things spread out from where we have lots of it towards where we have less, until it's in equilibrium
  - Heat
  - Gases
  - Water pollution
  - *etc.*
- Two effects contribute to this:
  - *Diffusion* is the thing's own tendency to level out
  - *Advection* is how the medium it is in carries it along when moving
- Advection works much faster than diffusion
  - Try to heat your apartment with and without a fan next to the heating element if you don't believe me

Advection terms

- Let's just think in 1 dimension to start with
- There are three parts to the equation:
  - ***U*** is the amount of whatever is spreading out
  - ***t*** is the time axis
  - ***x*** is the space axis
- If we're standing in some position along x,
  - dU / dt is the difference in how much U remains when time passes
  - dU / dx is the difference in how much U there is to our left and right
  - **v** is how quickly the medium is moving, and thus transporting some of our U in either direction

NTNU – Trondheim
Norwegian University of
Science and Technology

The advection equation

- Here it is: $$\frac{\delta U}{\delta t} = -v \cdot \frac{\delta U}{\delta x}$$
- Intuitively,
  - dU/dx indicates whether we find more or less U in a direction
  - v scales how quickly it will move from more to less
  - dU/dt is how much the amount of U will have changed in a moment

  - This thing is simple enough that we *could* just integrate it by hand and obtain a function U(t,x), which would solve the whole problem
  - That wouldn't give us anything to compute, though, so we'll cut it up in a way that is usually reserved for more complicated problems

Taylor polynomials

- Recall that $f(x+\Delta x) = f(x) + \frac{\Delta x}{1!} f'(x) + \frac{\Delta x^2}{2!} f''(x) + \ldots$
- The terms in a Taylor series shrink as the factorial of the term's index, so they rapidly become very small
- If we cut it off at the 2nd term, we'll only be making a small error, so we can use

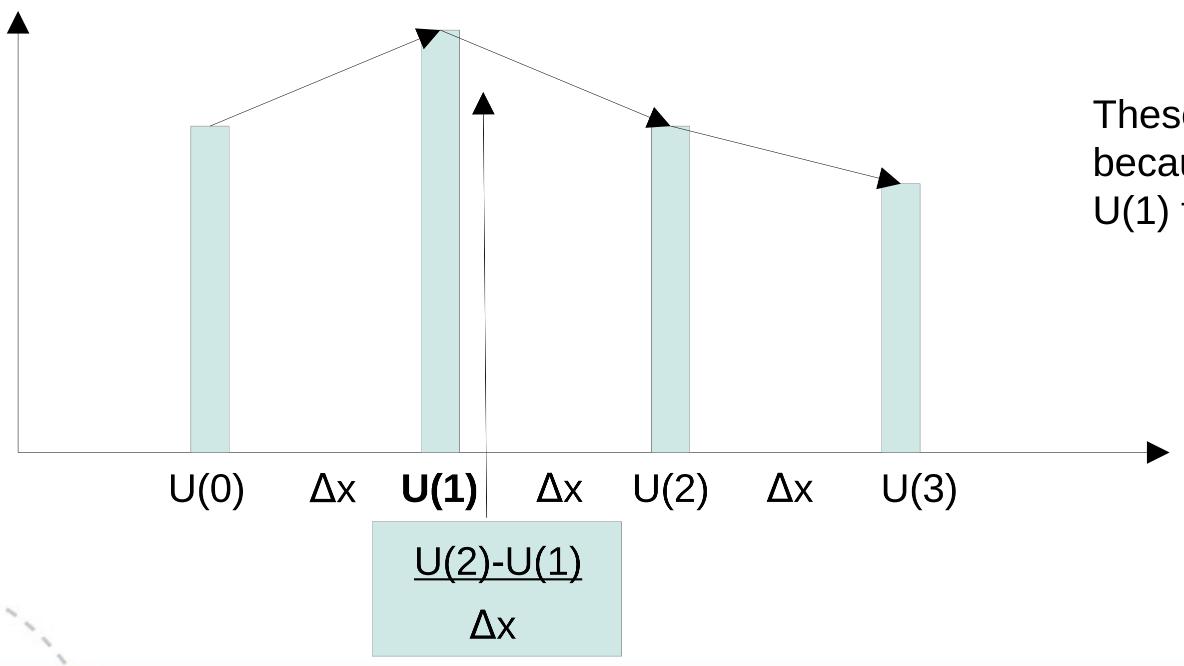
$$f(x+\Delta x) = f(x) + \Delta x f'(x)$$

and solve it for f'(x):

$$\frac{f(x+\Delta x) - f(x)}{\Delta x} = f'(x)$$

Discretizing dU/dx
(looking ahead)

- If we split our space axis into chunks of Δx, our expression gives the gradient of a straight line connecting the U-values in neighboring points:

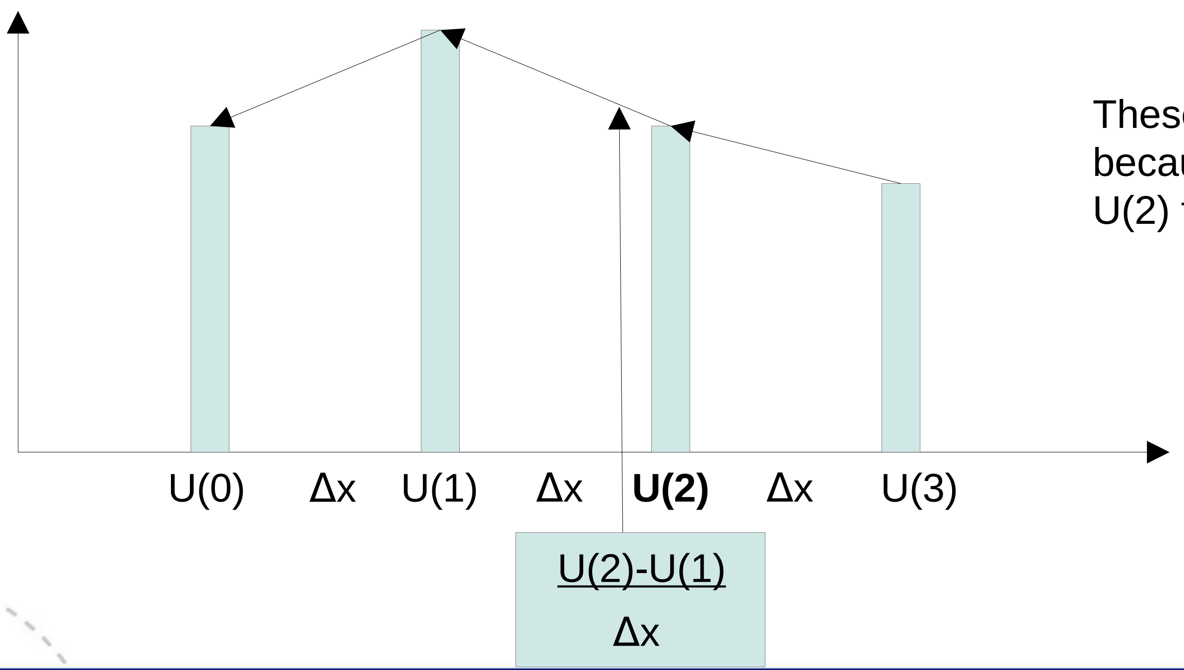


These are *forward differences,* because we find the gradient at U(1) from U(2) and U(1)

Discretizing dU/dx

(looking behind)

- If we split our space axis into chunks of Δx, our expression gives the gradient of a straight line connecting the U-values in neighboring points:

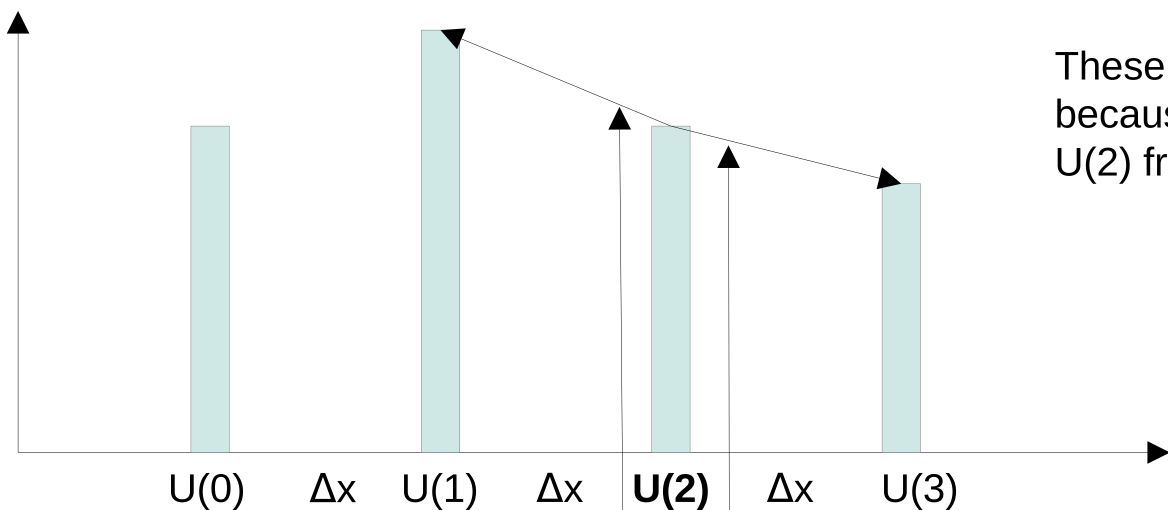


These are *backward differences*, because we find the gradient at U(2) from U(2) and U(1)

Discretizing dU/dx

(looking in both directions)

- If we average the forward and backward estimates, we get another estimate of the gradient at U(2):



These are *central differences*, because we find the gradient at U(2) from U(3), U(2), and U(1)

$$\frac{\dfrac{U(2)-U(1)}{\Delta x} + \dfrac{U(3)-U(2)}{\Delta x}}{2}$$

Simplifying the expression

- We can clean it up a little:

  (for brevity, let U_i be the U-value at the i-th step)

$$\frac{\frac{U_{i+1}-U_i}{\Delta x} + \frac{U_i - U_{i-1}}{\Delta x}}{2} = \frac{U_{i+1} - U_{i-1}}{2\Delta x}$$

- If we substitute that as our estimate of dU/dx, the overall equation becomes

$$\frac{\delta U}{\delta t} = -v \cdot \frac{U_{i+1} - U_{i-1}}{2\Delta x}$$

Discretizing dU/dt

(looking forward)

- If we similarly chop up the time axis in chunks of Δt, we can do the same thing all over again:

$$U'(t) = \frac{U(t+\Delta t) - U(t)}{\Delta t}$$

or with U^k as the U-value at the k-th step,

$$\frac{\delta U}{\delta t} = \frac{U^{k+1} - U^k}{\Delta t}$$

All together now

- The whole advection equation has now become

$$\frac{U_i^{k+1} - U_i^k}{\Delta t} = -v \cdot \frac{U_{i+1} - U_{i-1}}{2\Delta x}$$

which we can rearrange to obtain

$$U_i^{k+1} = U_i^k - v \cdot \frac{\Delta t}{2\Delta x} \cdot (U_{i+1}^k - U_{i-1}^k)$$

  – On the left hand side, we have U for the next moment in time
  – On the right hand side, we have only U values for the present moment in time
  – If we start from a distribution of U-s in one moment, we can
    - Calculate what it'll be in the next moment,
    - use the answer to calculate what it'll be in two moments,
    - use *that* answer to calculate… I'm sure you see where this is going

There’s one major missing piece

- The right hand side requires U-values from left and right neighbor points
- What about the very first and the very last U-value?
- We need two extra values that are determined in some other way, a.k.a. *boundary conditions*
- Three popular choices:
  - **Dirichlet** (say that the outside points equal some constants)
  - **Neumann** (say that the outside points mirror the inside points)
  - **Periodic** (say that the last outside point equals the first inside and vice versa, wrapping the domain around itself)
- We’ll do periodic boundaries today

There's one *minor* missing piece

- Our formula has a stability issue

$$U_i^{k+1} = U_i^k - v \cdot \frac{\Delta t}{2\Delta x} \cdot (U_{i+1}^k - U_{i-1}^k)$$

  - We're making a small rounding error for every time step, so sooner or later the answer will consist mostly of error
  - If we replace the U^k_i term on the r.h.s. with the average of its neighbors instead, we add some artificial inertia/friction/viscosity
  - That way, we get a movement that dies out instead of one that ultimately goes completely bananas:

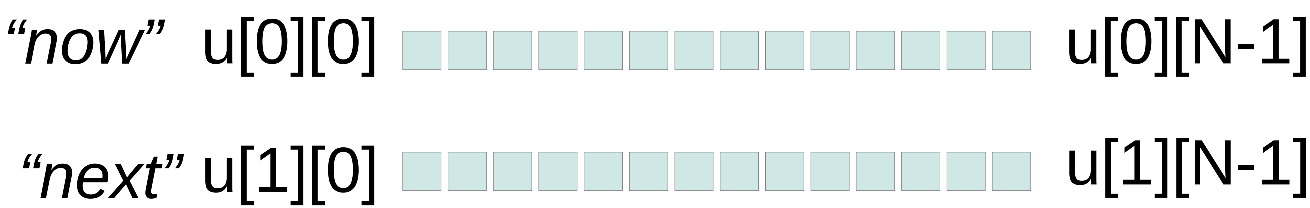
$$U_i^{k+1} = \frac{U_{i+1}^k + U_{i-1}^k}{2} - v \cdot \frac{\Delta t}{2\Delta x} \cdot (U_{i+1}^k - U_{i-1}^k)$$

NTNU – Trondheim
Norwegian University of Science and Technology

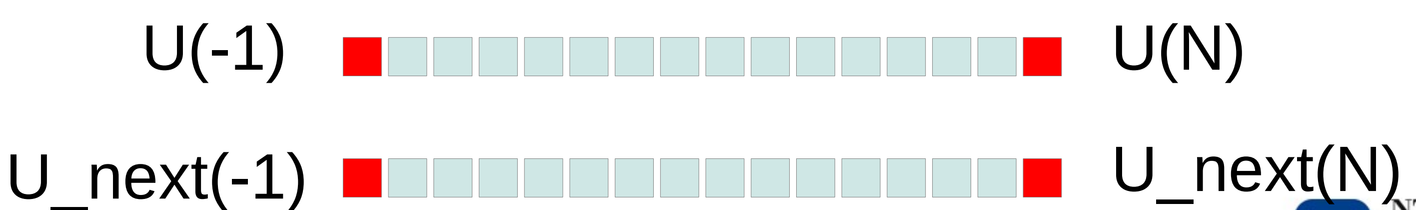
(This is called a "Lax-Friedrichs" method, but you don't have to remember that)

Turning it into software

- We'll certainly need two arrays of U-values, with N elements each:



- Give them extra boundary elements, and fix the indexing with a macro, for readability



The main stages of the program

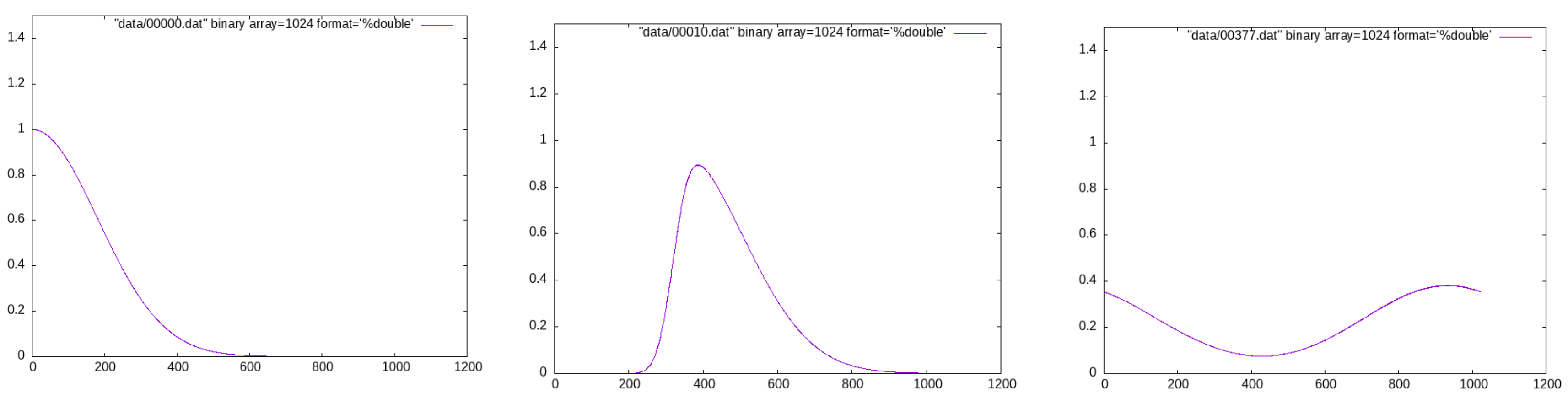
- Initialize:
  - Determine the number of points (N), the size of the x-axis (x_range), velocity (v), space step (dx), time step (dt), number of time steps to calculate (max_iter), and allocate the arrays.
  - Fill them with some interesting U-values we can move around
- Integrate in a loop
  - Copy the first/last values into the last/first boundary elements
  - Apply the formula we worked out to all the other elements
  - Switch the next-array into the now-array, and recycle the now-array as a place to write a new step
  - Repeat
- Finalize:
  - Delete all the allocations and stop

Seeing something happen

- If we just quietly worked out all the numbers, there would be no way to tell what the solution is at the end
- There's a variable 'snapshot_freq' in the code, which triggers the program to write all the numbers in its U-buffer into a file
- When we have a bunch of those files, we can plot their contents in graphs, and get a visual confirmation of how our function evolved over time
  - I like to use 'gnuplot' for that kind of thing

Seeing something happen

- The hard-coded initial state consists of a large displacement on the left
- It has a speed towards the right, and dies out over time

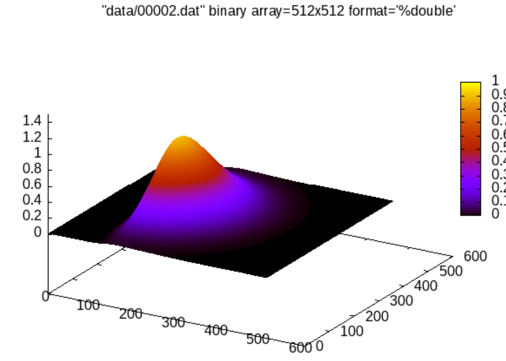


NTNU – Trondheim
Norwegian University of Science and Technology

This is terrible software design

- I know, right?
  - It’s not meant to be maintained
- The variables don’t have descriptive names
  - They’re chosen to resemble the terms in our expressions, so that they will be easy to recognize
- The whole program state is global
  - We don’t really need any modularity/encapsulation, the whole thing consists of fewer than 100 lines that only manipulate 2 arrays
- Everything is one long main()-function
  - I wanted it to be readable from top to bottom without having to skip around.
  - You can split it into sensibly named sub-functions if you wish

There's also a 2D version



- It's easy to make
  - Just add a y-velocity and a dU/dy term to the equation, discretize it in exactly the same way as for x, and make the U-arrays two-dimensional
  - Mind the boundary elements in the indexing macros
- It takes substantially longer to run
  - 1D is easier to explain, but it doesn't really give us a lot of work to parallelize

NTNU – Trondheim
Norwegian University of Science and Technology

Going forward

- Having covered how this program operates, I plan to return to it and make changes in order to illustrate things later on
- We can both benefit if you take it home, run it a few times, experiment with changing parts of it, *etc.*
  - **You** will be familiar with what it basically does when we create different variations
  - **I** will not have to go through a new set of greek letters every other week :)